Part 4

Taking OAuth further

In this section, we step back from the core OAuth protocol and look at the world of extensions, profiles, and complementary components that are built around that solid core. These include token formats, token management, client registration, user authentication, vertical-specific profiling, and proof of possession tokens. If you want to learn a bit about OpenID Connect, UMA, or PoP, this is where we get into it. Each of these particular topics could probably be a book unto themselves, but we're hoping to give you enough information to get you started.

11 OAuth tokens

This chapter covers

- What an OAuth token is
- Including information in structured JSON Web Tokens (JWT)
- Protecting token data with JOSE
- Looking up token information in real time with token introspection
- Managing a token lifecycle with token revocation

For all its redirects and flows and components, the OAuth protocol is ultimately about tokens. Think back to our cloud-printing example from chapter 1. In order for the photo-storage service to know that the printer had access to the photos, the printer service needed to give something to prove that authorization. We call the thing that the printer gives to the storage service an *access token*, and we've already been working with them extensively throughout the book. Now we're going to take a more in-depth look at OAuth tokens and managing them in an OAuth ecosystem.

11.1 What are OAuth tokens?

Tokens are at the core of all OAuth transactions. Clients fetch tokens from the authorization server to give to the protected resource. The authorization server creates tokens and hands them out to clients, managing resource owner delegations

Figure 11.1 The unofficial OAuth logo, modeled after a bus token

and client permissions to attach to the tokens along the way. The protected resource receives tokens from the clients and validates them, matching the attached permissions and rights to the request made by the client.

The token represents the results of the delegation act: a tuple of the resource owner, client, authorization server, protected resource, scopes, and everything else about the authorization decision. When a client needs to refresh its access tokens without bothering the resource owner again, it does this using another token: the refresh token. The OAuth token is the key mechanism that's at the center of OAuth's entire ecosystem, and without tokens there is arguably no OAuth. Even OAuth's unofficial logo is based on the look of a physical bus token (figure 11.1).

Yet for all of this focus on tokens, OAuth makes absolutely no claim or mention about what the content of a token is. As we've discussed in chapters 2 and 3, the client in OAuth doesn't need to know anything about the token itself. All the client needs to understand is how to get a token from the authorization server and how to use the token at the resource server. However, the authorization server and resource server do need to know about the token. The authorization server needs to know how to build a token to give to the client, and the resource server needs to know how to recognize and validate the token that's handed to it by the client.

Why would the OAuth core specifications leave out something so fundamental? By not specifying the tokens themselves, OAuth can be used in a wide variety of deployments with different characteristics, risk profiles, and requirements. OAuth tokens can expire, or be revocable, or be indefinite, or be some combination of these depending on circumstances. They can represent specific users, or all users in a system, or no users at all. They can have an internal structure, or be random nonsense, or be cryptographically protected, or even be some combination of these options. This flexibility and modularity allows OAuth to be adapted in ways that are difficult for more comprehensive security protocols such as WS-*, SAML, and Kerberos that do specify the token format and require all parties in the system to understand it.

Still, there are several common techniques for creating tokens and validating them, each of which has its benefits and drawbacks that make it applicable in different situations. In the exercises from chapters 3, 4, and 5 of this book, we created tokens that were random blobs of alphanumeric characters. They looked something like this on the wire:

```
s9nR4qv7qVadTUssVD5DqA7oRLJ2xonn
```

When the authorization server created the token, it stored the token's value in a shared database on disk. When the protected resource received a token from the client, it looked up the token's value in that same database to figure out what the token was good for. These tokens carry no information inside of them and instead act as simple

handles for data lookup. This is a perfectly valid and not uncommon method of creating and managing access tokens, and it has the benefit of being able to keep the token itself small while still providing a large amount of entropy.

It's not always practical to share a database between the authorization server and protected resource, especially when a single authorization server is protecting several different protected resources downstream. What then can we do instead? We're going to look at two other common options in this chapter: structured tokens and token introspection.

11.2 Structured tokens: JSON Web Token (JWT)

Instead of requiring a lookup into a shared database, what if we could create a token that had all of the necessary information inside of it? This way, an authorization server can communicate to a protected resource indirectly through the token itself, without any use of a network API call.

With this method, the authorization server packs in whatever information the protected resource will need to know, such as the expiration timestamp of the token and the user who authorized it. All of this gets sent to the client, but the client doesn't notice it because the token remains opaque to the client in all OAuth 2.0 systems. Once the client has the token, it sends the token to the protected resource as it would a random blob. The protected resource, which does need to understand the token, parses the information contained in the token and makes its authorization decisions based on that.

11.2.1 The structure of a JWT

To create this kind of token, we'll need a way to structure and serialize the information we want to carry. The JSON Web Token[1] format, or JWT,[2] provides a simple way to carry the kinds of information that we'd need to send with a token. At its core, a JWT is a JSON object that's wrapped into a format for transmission across the wire. The simplest form of JWT, an unsigned token, looks something like this:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
```

This probably looks like it's just as much of a random blob as the tokens that we were using before, but there's a lot more going on here. First, notice that there are two sections of characters separated by single periods. Each of these is a different part of the token, and if we split the token string on the dot character, we can process the sections separately. (A third section is implied after that last dot in our example, but we'll cover that in section 11.3.)

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0
.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
.
```

[1] RFC 7519 https://tools.ietf.org/html/rfc7519

[2] Commonly pronounced "jot."

Each value between the dots isn't random but is a Base64URL-encoded JSON object.[3] If we decode the Base64 and parse the JSON object inside the first section, we get a simple object.

```
{
  "typ": "JWT",
  "alg": "none"
}
```

Why Base64?

Why do we go through all this trouble of encoding things in Base64? After all, it's not human readable and it requires extra processing steps to make sense of it. Wouldn't it be better to use JSON directly? Part of the answer comes from the places where a JWT will typically find itself: in HTTP headers, query parameters, form parameters, and strings in various databases and programming languages. Each of these locations tends to have a limited set of characters that can be used without additional encoding. For example, in order to send a JSON object over an HTTP form parameter, the opening and closing brackets `{` and `}` would need to be encoded as `%7B` and `%7D`, respectively. Quotation marks, colons, and other common characters would also need to be encoded to their appropriate entity codes. Even something as common as the space character could be encoded as either `%20` or `+`, depending on the location of the token. Additionally, in many cases, the `%` character used for encoding itself needs to be encoded, often leading to accidental double-encoding of the values.

By natively using the Base64URL encoding scheme, JWT can be placed safely in any of these common locations without any additional encoding. Furthermore, since the JSON objects are coming through as an encoded string, they're less likely to be processed and reserialized by processing middleware, which we'll see is important in the next section. This kind of transportation-resistant armor is attractive to deployments and developers, and it's helped JWT find a foothold where other security token formats have faltered.

This header is always a JSON object and it's used to describe information about the rest of the token. The `typ` header tells the application processing the rest of the token what to expect in the second section, the payload. In our example, we're told that it's a JWT. Although there are other data containers that can use this same structure, JWT is far and away the most common and the best fit for our purposes as an OAuth token. This also includes the `alg` header with the special value `none` to indicate that this is an unsigned token.

The second section is the payload of the token itself, and it's serialized in the same way as the header: Base64URL-encoded JSON. Because this is a JWT, the payload can be any JSON object, and in our previous example it's a simple set of user data.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

[3] Specifically, it's Base64 encoding with a URL-safe alphabet and no padding characters.

11.2.2 JWT claims

In addition to a general data structure, JWT also gives us a set of claims for use across different applications. Although a JWT can contain any valid JSON data, these claims provide support for common operations that involve these kinds of tokens. All of these fields are optional in a JWT, but specific services are allowed to define their own inclusion requirements (table 11.1).

We can also add any additional fields that we need for our specific application. In our previous example token, we've added the `name` and `admin` fields to the payload, giving us a display name for the user as well as a boolean field that indicates whether this user is an administrator. The values of these fields can be any valid JSON value, including strings, numbers, arrays, or even other objects.

Table 11.1 Standard JSON web token claims

Claim Name	Claim Description
`iss`	The *issuer* of the token. This is an indicator of *who created this token*, and in many OAuth deployments this is the URL of the authorization server. This claim is a single string.
`sub`	The *subject* of the token. This is an indicator of *who the token is about*, and in many OAuth deployments this is a unique identifier for the resource owner. In most cases, the subject needs to be unique only within the scope of the issuer. This claim is a single string.
`aud`	The *audience* of the token. This is an indicator of *who is supposed to accept the token*, and in many OAuth deployments this includes the URI of the protected resource or protected resources that the token can be sent to. This claim can be either an array of strings or, if there's only one value, a single string with no array wrapping it.
`exp`	The *expiration* timestamp of the token. This is an indicator of *when the token will expire*, for deployments where the token will expire on its own. This claim is an integer of the number of seconds since the UNIX Epoch, midnight on January 1, 1970, in the Greenwich Mean Time (GMT) time zone.
`nbf`	The *not-before* timestamp of the token. This is an indicator of *when the token will begin to be valid*, for deployments where the token could be issued before it becomes valid. This claim is an integer of the number of seconds since the UNIX Epoch, midnight on January 1, 1970, in the GMT time zone.
`iat`	The *issued-at* timestamp of the token. This is an indicator of *when the token was created*, and is commonly the system timestamp of the issuer at the time of token creation. This claim is an integer of the number of seconds since the UNIX Epoch, midnight on January 1, 1970, in the GMT time zone.
`jti`	The *unique identifier* of the token. This is a value *unique to each token created by the issuer*, and it's often a cryptographically random value in order to prevent collisions. This value is also useful for preventing token guessing and replay attacks by adding a component of randomized entropy to the structured token that would not be available to an attacker.

The names of these fields can be any valid JSON string, which is true for any other JSON object, though the JWT specification[4] does have some guidance for avoiding collisions between JWT implementations. These guidelines are particularly useful if the JWT is intended to be consumed across security domains, where different claims can be defined and potentially have different semantics.

11.2.3 Implementing JWT in our servers

Let's add JWT support to our authorization server. Open up `ch-11-ex-1` and edit the `authorizationServer.js` file. In chapter 5 we created a server that issued unstructured, randomized tokens. We'll be modifying the server to produce unsigned JWT-formatted tokens here. Although we do recommend using a JWT library in practice, we'll be producing our JWTs by hand so that you can get a feel for what goes into these tokens. You'll get to play with a JWT library a little bit more in the next section.

First, locate the part of the code that generates the token itself. We'll be doing all of our coding in this section, starting by commenting out (or deleting) the following line:

```
var access_token = randomstring.generate();
```

To create our JWT, we first need a header. As in our previous example token, we're going to indicate that this token is a JWT and that it's unsigned. Since every token coming from our server will have the same characteristics, we can use a static object here.

```
var header = { 'typ': 'JWT', 'alg': 'none' };
```

Next, we're going to create an object to house the payload for our JWT and assign its fields based on what we care about in our token. We'll set the same issuer for every token to the URL of our authorization server, and we'll use the user variable from our authorization page as the subject of the token, if it exists. We'll also set the audience of the token to the URL of our protected resource. We'll mark a timestamp for the time at which the token was issued and set the token to expire five minutes in the future. Note that JavaScript natively deals with timestamps in milliseconds, whereas the JWT specification requires everything to be in seconds. Thus we have to account for a factor of 1000 when converting to and from the native values. Finally, we'll add in a randomized identifier for our token using the same random string generation function that we originally used to generate the whole token value. All together, creating our payload looks like the following:

```
var payload = {
   iss: 'http://localhost:9001/',
   sub: code.user ? code.user.sub : undefined,
   aud: 'http://localhost:9002/',
   iat: Math.floor(Date.now() / 1000),
   exp: Math.floor(Date.now() / 1000) + (5 * 60),
   jti: randomstring.generate(8)
};
```

[4] RFC 7519 https://tools.ietf.org/html/rfc7519

This will give us an object that looks something like the following, although of course the timestamps and random string will be different:

```
{
   "iss": "http://localhost:9001/",
   "sub": "alice",
   "aud": "http://localhost:/9002/",
   "iat": 1440538696,
   "exp": 1440538996,
   "jti": "Sl66JdkQ"
}
```

We can then take our header and payload objects, serialize the JSON as a string, encode that string using Base64URL encoding, and concatenate them together using periods as separators. We don't need to do anything special to the JSON objects as we serialize them, no special formatting or ordering of the fields, and any standard JSON serialization function will do.

```
var access_token = base64url.encode(JSON.stringify(header))
   + '.'
   + base64url.encode(JSON.stringify(payload))
   + '.';
```

Now our `access_token` value looks something like this unsigned JWT:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjkwMDEvI
   iwic3ViIjoiOVhFMy1KSTM0LTAwMTMyQSIsImF1ZCI6Imh0dHA6Ly9sb2NhbGhvc3Q6OTAwMi
   8iLCJpYXQiOjE0NjcyNDk3NzQsImV4cCI6MTQ2NzI1MDA3NCwianRpIjoiMFgyd2lQanUifQ.
```

Notice that our token now has an expiration associated with it, but the client doesn't have to do anything special with that change. The client can keep using the token until it stops working, at which point the client will go get another token as usual. The authorization server is allowed to provide an expiration hint to the client using the `expires_in` field of the token response, but the client doesn't even have to do anything with *that* either, and most clients don't.

Now it's time to have our protected resource check the incoming token for its information instead of looking up the token value in a database. Open up `protectedResource.js` and find the code that processes the incoming token. First we need to parse the token by performing the opposite actions that the authorization server used to create it: we split it on the dot characters to get the different sections. Then we'll decode the second part, the payload, from Base64 URL and parse the result as a JSON object.

```
var tokenParts = inToken.split('.');
var payload = JSON.parse(base64url.decode(tokenParts[1]));
```

This gives us a native data structure that we can check in our application. We're going to make sure that the token is coming from the expected issuer, that its timestamps fit the right ranges, and that our resource server is the intended audience of the token. Although these kinds of checks are often strung together with boolean logic, we've

broken these out into individual `if` statements so that each check can be read more clearly and independently.

```
if (payload.iss == 'http://localhost:9001/') {
   if ((Array.isArray(payload.aud) && __.contains(payload.aud, 'http://
     localhost:9002/')) ||
                payload.aud == 'http://localhost:9002/') {
         var now = Math.floor(Date.now() / 1000);
         if (payload.iat <= now) {
                if (payload.exp >= now) {
                       req.access_token = payload;
                }
         }
   }
}
```

If all of those checks pass, we'll hand the token's parsed `payload` on to the rest of the application, which can make authorization decisions based on fields such as the subject, if it so chooses. This is analogous to loading the data stored by the authorization server in its database in the previous version of the application.

Remember, the payload of a JWT is a JSON object, which our protected resource can now access directly from the request object. From here it's up to the other handler functions to determine whether this particular token is good enough to serve the requests in question, as we did when the token was stored in the shared database. The attributes included in the token's body in our example don't say that much, but we could easily include information about the client, resource owner, scopes, or other information pertinent to the protected resource's decision.

We haven't had to change our client code at all, even though the tokens that are being issued are different from what were being issued before. This is all thanks to the tokens being opaque to the client, which is a key simplifying factor in OAuth 2.0. In fact, the authorization server could have picked many different kinds of token formats without any change to the client software.

It's good that we can now carry information in the token itself, but is that enough?

11.3 Cryptographic protection of tokens: JSON Object Signing and Encryption (JOSE)

At this point, we, the authors, feel that we need to confess that we have just made you, the reader, do *a terribly insecure thing*. You may have already picked up on this important omission and are probably wondering whether we're off our rockers. What did we leave out? Simply put, if the authorization server outputs a token that is not protected in any way, and the protected resource trusts what's inside that token without any other checks, then it's trivial for the client, which receives the token in plain text, to manipulate the content of the token before presenting it to the protected resource. A client could even make up its own token out of whole cloth without ever talking to the authorization server, and a naïve resource server would simply accept and process it.

Since we almost certainly do *not* want that to happen, we should add some protection to this token. Thankfully for us, there's a whole suite of specifications that tell us exactly how to do this: the JSON Object Signing and Encryption standards,[5] or JOSE.[6] This suite provides signatures (JSON Web Signatures, or JWS), encryption (JSON Web Encryption, or JWE), and even key storage formats (JSON Web Keys, or JWK) using JSON as the base data model. The unsigned JWT that we built by hand in the last section is merely a special case of an unsigned JWS object with a JSON-based payload. Although the details of JOSE could fill a book on its own, we're going to look at two common cases: symmetric signing and validation using the HMAC signature scheme and asymmetric signing and validation using the RSA signature scheme. We'll also be using JWK to store our public and private RSA keys.

To do the heavy cryptographic lifting, we're going to be using a JOSE library called JSRSASign. This library provides basic signing and key management capabilities, but it doesn't provide encryption. We'll leave encrypted tokens as an exercise for the reader.

11.3.1 Symmetric signatures using HS256

For our next exercise, we're going to sign our token using a shared secret at the authorization server and then validate that token using the shared secret at the protected resource. This is a useful approach when the authorization server and protected resources are tied closely enough to have a long-term shared secret, similar to an API key, but do not have a direct connection to each other to validate each token directly.

Open up `ch-11-ex-2` and edit the `authorizationServer.js` and `protectedResource.js` files for this exercise. First, we're going to add a shared secret to our authorization server. Toward the top of the file, find the variable definition for `sharedTokenSecret` and see that we've set it to a secret string. In a production environment, the secret is likely to be managed by some kind of credential management process, and its value isn't likely to be this short or this easy to type, but we've simplified things for the exercises.

```
var sharedTokenSecret = 'shared OAuth token secret!';
```

Now we'll use that secret to sign the token. Our code is set up like the last exercise, creating an unsigned token, so locate the token generation code to continue. We first need to change the header parameter to indicate that we're using the HS256 signature method.

```
var header = { 'typ': 'JWT', 'alg': 'HS256'};
```

Our JOSE library requires us to do the JSON serialization (but not the Base64 URL encoding) before passing data into the signing function, but we've already got that set

[5] JWS: RFC 7515 https://tools.ietf.org/html/rfc7515;
JWE: RFC 7516 https://tools.ietf.org/html/rfc7516;
JWK: RFC 7517 https://tools.ietf.org/html/rfc7517;
JWA: RFC 7518 https://tools.ietf.org/html/rfc7518

[6] Intended to be pronounced like the Spanish given name José, or "ho-zay."

up. This time, instead of concatenating the strings together with dots, we're going to use our JOSE library to apply the HMAC signature algorithm, using our shared secret, to the token. Due to a quirk in our chosen JOSE library, we need to pass in the shared secret as a hex string; other libraries will have different requirements for getting the keys in the right format. The output of the library will be a string that we'll use as the token value.

```
var access_token = jose.jws.JWS.sign(header.alg,
   JSON.stringify(header),
   JSON.stringify(payload),
   new Buffer(sharedTokenSecret).toString('hex'));
```

The final JWT looks something like the following:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjkwMDEv
   Iiwic3ViIjoiOVhFMy1KSTM0LTAwMTMyQSIsImF1ZCI6Imh0dHA6Ly9sb2NhbGhvc3Q6OTAwMi
   8iLCJpYXQiOjE0NjcyNTEwNzMsImV4cCI6MTQ2NzI1MTM3MywianRpIjoiaEZLUUpSNmUifQ.
   WqRsY03pYwuJTx-9pDQXftkcj7YbRn95o-16NHrVugg
```

The header and payload remain the same Base64URL-encoded JSON strings, as before. The signature is placed after the final dot in the JWT format, as a Base64URL-encoded set of bytes, making the overall structure `header.payload.signature` for a signed JWT. When we split the sections on the dots, it's a little easier to see the structure.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjkwMDEvIiwic3ViIjoiOVhFMy1KSTM0LTAwMTMyQSIs
   ImF1ZCI6Imh0dHA6Ly9sb2NhbGhvc3Q6OTAwMi8iLCJpYXQiOjE0NjcyNTEwNzMsImV4cCI6MT
   Q2NzI1MTM3MywianRpIjoiaEZLUUpSNmUifQ
.
WqRsY03pYwuJTx-9pDQXftkcj7YbRn95o-16NHrVugg
```

Now you can see that our unsigned JWT was a case of having an empty (missing) signature section. The rest of the server remains unchanged, as we're still storing the token in the database. However, if we wanted to, we could remove the storage requirement on our authorization server entirely because the token is recognizable by the server from its signature.

Once again, our client is none the wiser that the token format has changed. However, we're going to need to edit the protected resource so that it can check the token's signature. To do this, open `protectedResource.js` and note the same random secret string at the top of the file. Once again, in a production environment, this is likely handled through a key management process and the secret isn't likely to be this simple to type.

```
var sharedTokenSecret = 'shared OAuth token secret!';
```

First we need to parse the token, but that's pretty much like last time.

```
var tokenParts = inToken.split('.');
var header = JSON.parse(base64url.decode(tokenParts[0]));
var payload = JSON.parse(base64url.decode(tokenParts[1]));
```

Notice that we'll be using the token header this time. Next, verify the signature based on our shared secret, and that will be our first check of the token's contents. Remember, our library needs the secret to be converted to hex before it can validate things.

```
if (jose.jws.JWS.verify(inToken,
        new Buffer(sharedTokenSecret).toString('hex'),
        [header.alg])) {
}
```

All of the previous token validity checks go inside this if statement

Take special note that we have passed in the token string exactly as it was handed to us on the wire. We didn't use the decoded or parsed JSON objects, nor did we re-encode them on our own. Had we done any of that, it's entirely possible (and completely legal) that the serialization of the JSON would have been slightly different, such as by adding or removing spaces and indentation or by re-ordering the members of a data object. As we discussed, the JOSE specifications effectively armor the token against transformation in transit specifically so that this validation step can be performed without any renormalization.

Only if the signature is valid do we parse the JWT and check its contents for consistency. If all checks pass, we can hand it off to the application, as we did previously. Now the resource server will only accept tokens that have been signed by the secret that it shares with the authorization server. To test this, edit the secret in either the authorization server's or protected resource's code so that they differ. The resource server should reject the resulting token.

11.3.2 Asymmetric signatures using RS256

In this section's exercise, we're once again going to sign the token with a secret key, as we did in the last section. However, this time, we're going to use public key cryptography to do it. With a shared secret, both systems need the same key either to create or to validate the signature. This effectively means that either the authorization server or the resource server could create the tokens in the last exercise, because they both had access to the keying material needed to do so. With public key cryptography, the authorization server has both a private key and a public key that it can use to generate tokens, whereas the protected resource needs to be able to access only the authorization server's public key to verify the token. Unlike with a shared secret, the protected resource has no way of generating its own valid tokens even though it can easily verify them. We're going to be using the `RS256` signature method from JOSE, which uses the RSA algorithm under the hood.

Open up `ch-11-ex-3` and start with the `authorizationServer.js` file. First, we need to add a public and private key pair to our authorization server. Our key pair is a 2048-bit RSA key, which is the minimum recommended size. We're using keys stored in the JSON-based JWK format for this exercise, and they can be read natively by our library. To keep you from having to type in this convoluted set of characters *exactly* as

it's written in the book, we've gone ahead and included it in the code for you, so go check it out.

```
var rsaKey = {
  "alg": "RS256",
  "d": "ZXFizvaQ0RzWRbMExStaS_-yVnjtSQ9YslYQF1kkuIoTwFuiEQ2OywBfuyXhTvVQxIiJq
    PNnUyZR6kXAhyj__wS_Px1EH8zv7BHVt1N5TjJGlubt1dhAFCZQmgz0D-PfmATdf6KLL4HIijG
    rE8iYOPYIPF_FL8ddaxx5rsziRRnkRMX_fIHxuSQVCe401hSS3QBZOgwVdWEb1JuODT7KUk7xPp
    MTw5RYCeUoCYTRQ_KO8_NQMURi3GLvbgQGQgk7fmDcug3MwutmWbpe58GoSCkmExUS0U-KEkH
    tFiC8L6fN2jXh1whPeRCa9eoIK8nsIY05gnLKxXTn5-aPQzSy6Q",
  "e": "AQAB",
  "n": "p8eP5gL1H_H9UNzCuQS-vNRVz3NWxZTHYk1tG9VpkfFjWNKG3MFTNZJ1l5g_COMm2_2i_
    YhQNH8MJ_nQ4exKMXrWJB4tyVZohovUxfw-eLgu1XQ8oYcVYW8ym6Um-BkqwwWL6CXZ70X81
    YyIMrnsGTyTV6M8gBPun8g2L8KbDbXR1lDfOOWiZ2ss1CRLrmNM-GRp3Gj-ECG7_3Nx9n_s5
    to2ZtwJ1GS1maGjrSZ9GRAYLrHhndrL_8ie_9DS2T-ML7QNQtNkg2RvLv4f0dpjRYI23djxV
    tAylYK4oiT_uEMgSkc4dxwKwGuBxSO0g9JOobgfy0--FUHHYtRi0dOFZw",
  "kty": "RSA",
  "kid": "authserver"
};
```

This key pair was randomly generated, and in a production environment you'll want to have a unique key for each service. As an added exercise, generate your own JWK using a JOSE library and replace the one in the code here.

Next we need to sign the token using our private key. The process is similar to how we handled the shared secret, and we'll be working in the token generation function again. First we need to indicate that our token is signed with the RS256 algorithm. We're also going to indicate that we're using the key with the key ID (`kid`) of `authserver` from our authorization server. The authorization server may have only one key right now, but if you were to add other keys to this set, you'd want the resource server to be able to know which one you used.

```
var header = { 'typ': 'JWT', 'alg': rsaKey.alg, 'kid': rsaKey.kid };
```

Next, we need to convert our JWK-formatted key pair into a form that our library can use for cryptographic operations. Thankfully, our library gives us a simple utility for doing that.[7] We can then use this key to sign the token.

```
var privateKey = jose.KEYUTIL.getKey(rsaKey);
```

Then we'll create our access token string much like we did before, except this time we use our private key and the RS256 asymmetric signing algorithm.

```
var access_token = jose.jws.JWS.sign(header.alg,
   JSON.stringify(header),
   JSON.stringify(payload),
   privateKey);
```

[7] Other libraries and other platforms may need to have key objects created from the different parts of the JWK.

The result is the token similar to the previous one, but it's now been signed asymmetrically.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImF1dGhzZXJ2ZXIifQ.eyJpc3MiOiJodH
   RwOi8vbG9jYWxob3N0OjkwMDEvIiwic3ViIjoiOVhFMy1KSTM0LTAwMTMyQSIsImF1ZCI6Imh
   0dHA6Ly9sb2NhbGhvc3Q6OTAwMi8iLCJpYXQiOjE0NjcyNTE5NjksImV4cCI6MTQ2NzI1MjI2
   OSwianRpIjoidURYMWNwVnYifQ.nK-tYidfd6IHW8iwJ1ZHcPPnbDdbjnveunKrpOihEb0JD5w
   fjXoYjpToXKfaSFPdpgbhy4ocnRAfKfX6tQfJuFQpZpKmtFG8OVtWpiOYlH4Ecoh3soSkaQyIy
   4L6p8o3gmgl9iyjLQj4B7Anfe6rwQlIQi79WTQwE9bd3tgqic5cPBFtPLqRJQluvjZerkSdUo
   7Kt8XdyGyfTAiyrsWoD1H0WGJm6IodTmSUOH7L08k-mGhUHmSkOgwGddrxLwLcMWWQ6ohmXa
   Vv_Vf-9yTC2STHOKuuUm2w_cRE1sF7JryiO7aFRa8JGEoUff2moaEuLG88weOT_S2EQBhYB
   0vQ8A
```

The header and payload are still Base64URL-encoded JSON, and the signature is a Base64URL-encoded array of bytes. The signature is much longer now as a result of using the RSA algorithm.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImF1dGhzZXJ2ZXIifQ
.
eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjkwMDEvIiwic3ViIjoiOVhFMy1KSTM0L
   TAwMTMyQSIsImF1ZCI6Imh0dHA6Ly9sb2NhbGhvc3Q6OTAwMi8iLCJpYXQiOjE0
   NjcyNTE5NjksImV4cCI6MTQ2NzI1MjI2OSwianRpIjoidURYMWNwVnYifQ
.
nK-tYidfd6IHW8iwJ1ZHcPPnbDdbjnveunKrpOihEb0JD5wfjXoYjpToXKfaSFPdpgbhy4ocnRAfK
   fX6tQfJuFQpZpKmtFG8OVtWpiOYlH4Ecoh3soSkaQyIy4L6p8o3gmgl9iyjLQj4B7Anfe6rwQlI
   Qi79WTQwE9bd3tgqic5cPBFtPLqRJQluvjZerkSdUo7Kt8XdyGyfTAiyrsWoD1H0WGJm6IodTmS
   UOH7L08k-mGhUHmSkOgwGddrxLwLcMWWQ6ohmXaVv_Vf-9yTC2STHOKuuUm2w_cRE1sF7JryiO7
   aFRa8JGEoUff2moaEuLG88weOT_S2EQBhYB0vQ8A
```

The client once again remains unchanged, but we do have to tell the protected resource how to validate the signature of this new JWT. Open up `protectedResource.js` so that we can tell it the server's public key. Once again, instead of making you painstakingly transcribe the key information, we've gone ahead and included it in the file for you.

```
var rsaKey = {
  "alg": "RS256",
  "e": "AQAB",
  "n": "p8eP5gL1H_H9UNzCuQS-vNRVz3NWxZTHYk1tG9VpkfFjWNKG3MFTNZJ1l5g_COMm2_2i_
    YhQNH8MJ_nQ4exKMXrWJB4tyVZohovUxfw-eLgu1XQ8oYcVYW8ym6Um-BkqwwWL6CXZ70X81
    YyIMrnsGTyTV6M8gBPun8g2L8KbDbXR1lDfOOWiZ2ss1CRLrmNM-GRp3Gj-ECG7_3Nx9n_s5
    to2ZtwJ1GS1maGjrSZ9GRAYLrHhndrL_8ie_9DS2T-ML7QNQtNkg2RvLv4f0dpjRYI23djxV
    tAylYK4oiT_uEMgSkc4dxwKwGuBxSO0g9JOobgfy0--FUHHYtRi0dOFZw",
  "kty": "RSA",
  "kid": "authserver"
};
```

This data is from the same key pair as the one in the authorization server, but it doesn't contain the private key information (represented by the `d` element in an RSA key). The effect is that the protected resource can only verify incoming signed JWTs, but it cannot create them.

Do I have to copy my keys all over the place?

You might think it's onerous to copy signing and verification keys between pieces of software like this, and you'd be right. If the authorization server ever decides to update its keys, all copies of the corresponding public key need to be updated in all protected resources downstream. For a large OAuth ecosystem, that can be problematic.

One common approach, used by the OpenID Connect protocol that we'll cover in chapter 13, is to have the authorization server publish its *public* key at a known URL. This will generally take the form of a JWK Set, which can contain multiple keys and looks something like this.

```
{
  "keys": [
    {
      "alg": "RS256",
      "e": "AQAB",
      "n": "p8eP5gL1H_H9UNzCuQS-vNRVz3NWxZTHYk1tG9VpkfFjWNKG3MFTNZJ1l5g_
COMm2_2i_YhQNH8MJ_nQ4exKMXrWJB4tyVZohovUxfw-eLgu1XQ8oYcVYW8ym6Um-Bkqww
WL6CXZ70X81YyIMrnsGTyTV6M8gBPun8g2L8KbDbXR1lDfOOWiZ2ss1CRLrmNM-GRp3Gj-
ECG7_3Nx9n_s5to2ZtwJ1GS1maGjrSZ9GRAYLrHhndrL_8ie_9DS2T-ML7QNQtNkg2RvLv4f
0dpjRYI23djxVtAylYK4oiT_uEMgSkc4dxwKwGuBxSO0g9JOobgfy0--FUHHYtRi0dOFZw",
      "kty": "RSA",
      "kid": "authserver"
    }
  ]
}
```

The protected resources can then fetch and cache this key as needed. This approach allows the authorization server to rotate its keys whenever it sees fit, or add new keys over time, and the changes will automatically propagate throughout the network.

For an added exercise, modify the server to publish its public key as a JWK set, and modify the protected resource to fetch this key over the network as needed. Be very careful that the authorization server publishes only its *public* key and not the *private* key as well!

Now we'll use our library to validate the signatures of incoming tokens based on the server's public key. Load up the public key into an object that our library can use, and then use that key to validate the token's signature.

```
var publicKey = jose.KEYUTIL.getKey(rsaKey);

if (jose.jws.JWS.verify(inToken,
       publicKey,
       [header.alg])) {
}
```

All of the previous token validity checks go inside this if statement

We'll still need to perform all the same checks against the token that we performed when we had an unsigned token. The payload object is once again handed off to

the rest of the application to allow it to decide whether the presented token is sufficient for the given request. Now that this has been set up, the authorization server can choose to include additional information for the protected resource's consumption, such as scopes or client identifiers. For an additional exercise, add some of this information in using your own JWT claims and have the protected resource reach to their values.

11.3.3 Other token protection options

The methods that we've gone over in these exercises aren't the only JOSE-based means of protecting a token's contents. For instance, we used the HS256 symmetric signature method previously, which provides a 256-byte hash of the token's contents. JOSE also defines HS384 and HS512, both of which use larger hashes to provide increased security at the cost of larger token signatures. Similarly, we used the RS256 asymmetric signature method, which provides a 256-byte hash of the RSA signature output. JOSE also defines the RS384 and RS512 methods, with the same kinds of trade-offs as their symmetric counterparts. JOSE also defines the PS256, PS384, and PS512 signature methods, all based on a different RSA signature and hashing mechanism.

JOSE also provides elliptical curve support, with the core standard referencing three curves and associated hashes in ES256, ES384, and ES512. Elliptical curve cryptography has several benefits over RSA cryptography, including smaller signature sizes and lower processing requirements for validation, but support for the underlying cryptographic functions isn't nearly as widespread as RSA at the time of this writing. On top of this, JOSE's list of algorithms is extensible by new specifications, allowing new algorithms to be defined as they're invented and needed.

Sometimes signatures aren't enough, though. With a token that's only signed, the client could potentially peek at the token itself and find out things that it might not be privileged to know, such as a user identifier in the `sub` field. The good news is that in addition to signatures, JOSE provides an encryption mechanism called JWE with several different options and algorithms. Instead of a three-part structure, a JWT encrypted with JWE is a five-part structure. Each portion still uses Base64 URL encoding, and the payload is now an encrypted object that can't be read without access to the appropriate key. Covering the JWE process is a bit much for this chapter, but for an advanced exercise try adding JWE to the tokens. First, give the resource server a key pair and give the authorization server access to the public key portion of this key pair. Then use the public key to encrypt the contents of the token using JWE. Finally, have the resource server decrypt the contents of the token using its own private key and pass the payload of the token along to the application.

Let's get COSE[8]

An emerging standard called CBOR Object Signing and Encryption (COSE) provides much of the same functionality of JOSE but based on the Concise Binary Object Representation (CBOR) data serialization. As its name suggests, CBOR is a non-human-readable binary format that's designed for environments in which space is at a premium. Its underlying data model is based on JSON, and anything represented in JSON can easily be translated into CBOR. The COSE specification is attempting to do for CBOR what JOSE did for JSON before it, which means that it will likely become a viable option for compact JWT-like tokens in the near future.

11.4 Looking up a token's information online: token introspection

Packing the information about the token into the token itself does have its drawbacks. The tokens themselves can grow to be quite large as they incorporate all the required claims and cryptographic structures required to protect those claims. Furthermore, if the protected resource is relying solely on information carried in the token itself, it becomes prohibitively difficult to revoke active tokens once they've been created and sent into the wild.

11.4.1 The introspection protocol

The OAuth Token Introspection protocol[9] defines a mechanism for a protected resource to actively query an authorization server about the state of the token. Since the authorization server minted the token, it's in the perfect position to know the surrounding details of the authorization delegation that the token represents.

The protocol is a simple augmentation of OAuth. The authorization server issues the token to the client, the client presents the token to the protected resource, and the protected resource introspects the token at the authorization server (figure 11.2).

The introspection request is a form-encoded HTTP request to the authorization server's introspection endpoint, which allows the protected resource to ask, "Someone gave me this token; what is it good for?" of the authorization server. The protected resource authenticates itself during this request so that the authorization server can tell who is asking the question and potentially give a different response depending on who's asking. The introspection specification doesn't dictate *how* the protected resource needs to authenticate itself, only that it does so. In our example, the protected resource authenticates using an ID and secret over HTTP Basic, much in the same way that an OAuth client would authenticate itself to the token endpoint. This could also be accomplished using a separate access token, which is how the UMA protocol discussed in chapter 14 does it.

[8] Pronounced "cozy", as in "a cozy couch."
[9] RFC 7662 https://tools.ietf.org/html/rfc7662

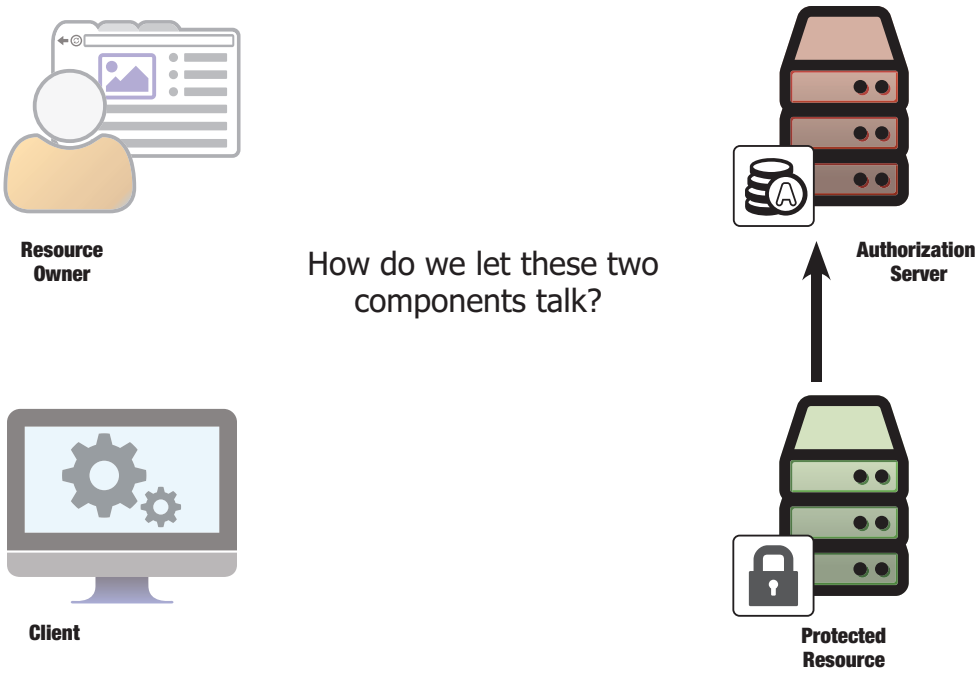

Figure 11.2 Connecting the protected resource to the authorization server

```
POST /introspect HTTP/1.1
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic
   cHJvdGVjdGVkLXJlc291cmNlLTE6cHJvdGVjdGVkLXJlc291cmNlLXNlY3JldC0x

token=987tghjkiu6trfghjuytrghj
```

The introspection response is a JSON document that describes the token. Its contents are similar to the payload of a JWT, and any valid JWT claim can be used as part of the response.

```
HTTP 200 OK
Content-type: application/json

{
  "active": true,
  "scope": "foo bar baz",
  "client_id": "oauth-client-1",
  "username": "alice",
  "iss": "http://localhost:9001/",
  "sub": "alice",
  "aud": "http://localhost:/9002/",
  "iat": 1440538696,
  "exp": 1440538996,
}
```

The introspection specification also defines several claims in addition to those defined by JWT, the most important of which is the `active` claim. This claim tells the protected resource whether the current token is active at the authorization server, and it's the only claim required to be returned. Because there are many different kinds of OAuth token deployments, there is no single definition of what an active token is. In general, though, this means that the token was issued by this authorization server, it hasn't expired yet, it hasn't been revoked, and the requesting protected resource is allowed to fetch information about it. Interestingly, this piece of information is something that can't be included in the token itself, since no token would ever claim to be non-active.

The introspection response may also include the scope of the token, represented as a space-separated list of scope strings as in the original OAuth request. As we've seen in chapter 4, the scope of a token allows a protected resource to determine the rights delegated to the client by the resource owner in a more fine-grained manner. Finally, information about the client and user can also be included. All of this together can give the protected resource a rich set of data from which to make the final authorization decision.

The use of token introspection incurs the overhead of increased network traffic on the OAuth systems. To combat this, the protected resource is allowed to cache the results of the introspection call for a given token. It's recommended that the cache be short lived relative to the expected lifetime of the token in order to limit the chance of a token being revoked while the cache is in effect.

11.4.2 Building the introspection endpoint

Now we're going to build introspection support into our applications. Open up `ch-11-ex-4` and start with the `authorizationServer.js` where we're going to build the introspection endpoint. First, we'll add some credentials for our protected resource that will allow it to authenticate to our introspection endpoint.

```
var protectedResources = [
   {
        "resource_id": "protected-resource-1",
        "resource_secret": "protected-resource-secret-1"
   }
];
```

We're deliberately modeling this off client authentication, which is one of the default options given by the introspection specification for protected resource authentication. We've included a `getProtectedResource` function to mirror the `getClient` function we created in chapter 5.

```
var getProtectedResource = function(resourceId) {
   return __.find(protectedResources, function(protectedResource) { return
   protectedResource.resource_id == resourceId; });
};
```

We're going to host our introspection endpoint at `/introspect` on our authorization server, and it's going to listen for POST requests.

```
app.post('/introspect', function(req, res) {

});
```

Our protected resource is going to authenticate using HTTP Basic authentication and a shared secret, so we'll look for that in the `Authorization` header as we do for client credentials at the token endpoint.

```
var auth = req.headers['authorization'];
var resourceCredentials = decodeClientCredentials(auth);
var resourceId = resourceCredentials.id;
var resourceSecret = resourceCredentials.secret;
```

Once we have the presented credentials, look up the resource using our helper function and figure out whether or not the secret matches.

```
var resource = getProtectedResource(resourceId);
if (!resource) {
   res.status(401).end();
   return;
}

if (resource.resource_secret != resourceSecret) {
   res.status(401).end();
   return;
}
```

Now we have to look up the token in our database. If we find the token, we're going to add all the information that we have about the token to our response and send it back as a JSON object. If we don't find the token, we send back only a notification that the token was not active.

```
var inToken = req.body.token;
nosql.one(function(token) {
   if (token.access_token == inToken) {
        return token;
   }
}, function(err, token) {
   if (token) {

        var introspectionResponse = {
                active: true,
                iss: 'http://localhost:9001/',
                aud: 'http://localhost:9002/',
                sub: token.user ? token.user.sub : undefined,
                username: token.user ? token.user.preferred_username : undefined,
                scope: token.scope ? token.scope.join(' ') : undefined,
                client_id: token.client_id
        };

        res.status(200).json(introspectionResponse);
        return;
   } else {
        var introspectionResponse = {
                active: false
```

```
        };
        res.status(200).json(introspectionResponse);
        return;
    }
});
```

For security reasons, it's important that we don't tell the protected resource exactly why the token wasn't active—whether it expired, was revoked, or was never issued in the first place—but instead say that it wasn't any good. Otherwise, a compromised protected resource could be used by an attacker to fish about at the authorization server for information about tokens. For legitimate transactions, it ultimately doesn't matter *why* the token isn't good, only that it's not.

Bringing it all together, our introspection endpoint looks like listing 11 in appendix B. The introspection endpoint is supposed to look up refresh tokens as well, but we're leaving that additional functionality as an exercise to the reader.

11.4.3 Introspecting a token

Now that we have an introspection endpoint to call, we need to set up our protected resource to call it. We're going to continue in the same exercise from the last section, `ch-11-ex-4`, but now open up `protectedResource.js` and edit it. Start off by giving the protected resource its ID and secret, as we did with our client in chapter 5.

```
var protectedResource = {
    "resource_id": "protected-resource-1",
    "resource_secret": "protected-resource-secret-1"
};
```

Next, inside the `getAccessToken` function, we're going to call the introspection endpoint. This is a simple HTTP POST with the previous ID and secret sent as HTTP Basic parameters and the token value that was received from the client sent as a form parameter.

```
var form_data = qs.stringify({
    token: inToken
});
var headers = {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Authorization': 'Basic ' + encodeClientCredentials(protectedResource
      .resource_id, protectedResource.resource_secret)
};

var tokRes = request('POST', authServer.introspectionEndpoint, {
    body: form_data,
    headers: headers
});
```

Finally, take the response from the introspection endpoint and parse it as a JSON object. If the active claim comes back as true, then pass the result of the introspection call in to the rest of the application for further processing.

```
if (tokRes.statusCode >= 200 && tokRes.statusCode < 300) {
   var body = JSON.parse(tokRes.getBody());

   console.log('Got introspection response', body);
   var active = body.active;
   if (active) {
        req.access_token = body;
   }
}
```

From here, the protected resource serving functions will decide whether the token is sufficient or appropriate for the request at hand.

11.4.4 Combining introspection and JWT

In this chapter, we've presented structured tokens (specifically JWT) and token introspection as two alternative ways to carry information between the authorization server and the protected resource. It might seem as though you need to pick either one method or the other, but in truth, they can be used together to great effect.

The JWT can be used to carry core information, for example, the expiration, unique identifier, and issuer. These pieces of information are going to be needed by every protected resource to provide a first-order check of whether the token is trusted. From there, the protected resource can perform token introspection to determine more detailed (and potentially sensitive) information about the token, such as the user who authorized it, the client it was issued to, and the scopes it was issued with.

This approach is particularly useful in cases in which a protected resource is set up to accept access tokens from a variety of authorization servers. The protected resource can parse the JWT to find out which authorizations server issued the token and then introspect the token at the correct authorization server to find out more.

The state of the token

For a client, it doesn't matter if its token was revoked by another party, since OAuth clients always need to be ready to go get a new token. The OAuth protocol doesn't differentiate error responses based on whether the token was revoked, expired, or otherwise invalid, because the client's response is always the same.

However, as a protected resource, it's definitely important to know whether a token was revoked or not, because accepting a revoked token would be a huge security hole. Huge security holes are generally considered bad things to have. If the protected resource uses a local database lookup or a live query such as introspection, it can find out easily and quickly that a token was revoked. But what if it's using a JWT?

Since a JWT is ostensibly self-contained, it's considered *stateless*. There's no way to indicate to a protected resource that it has been revoked without resorting to external signals. The same problem occurs in a certificate-based public key infrastructure (PKI), in which a certificate is valid if all of the signatures match. The revocation problem was addressed here using *certificate revocation lists* and the *online certificate status protocol (OCSP)*, equivalent to token introspection in the OAuth world.

11.5 Managing the token lifecycle with token revocation

OAuth tokens usually follow a predictable lifecycle. They're created by the authorization server, used by the client, and validated by the protected resource. They might expire on their own or be revoked by the resource owner (or an administrator) at the authorization server. OAuth's core specifications provide mechanisms for getting and using tokens in a variety of ways, as we've seen. A refresh token even allows a client to request a new access token to replace an invalidated one. In sections 11.2 and 11.3, we've seen how JWT and token introspection can be used to help a protected resource validate a token. Sometimes, though, the client knows that it will no longer need a token. Does it need to wait for the token to expire, or for someone else to revoke it?

So far, we haven't seen any mechanism that would let the client tell the authorization server to revoke otherwise valid tokens, but this is where the OAuth Token Revocation specification[10] comes in. This specification allows clients to be proactive in managing token lifecycles in response to triggering events on the client side. For instance, the client may be a native application being uninstalled from the user's device, or it may provide a user interface that allows the user to deprovision the client. Perhaps, even, the client software has detected suspicious behavior and wants to limit the damage to the protected resources it has been authorized for. Whatever the triggering event, the token revocation specification lets the client signal the authorization server that the tokens it has issued should no longer be used.

11.5.1 The token revocation protocol

OAuth token revocation is a simple protocol that allows a client to say, succinctly, "I have this token and I want you to get rid of it" to an authorization server. Much like token introspection that we covered in section 11.4, the client makes an authenticated HTTP POST request to a special endpoint, the revocation endpoint, with the token to be revoked as a form-encoded parameter in the request body.

```
POST /revoke HTTP/1.1
Host: localhost:9001
Accept: application/json
Content-type: application/x-www-form-encoded
Authorization: Basic b2F1dGgtY2xpZW50LTE6b2F1dGgtY2xpZW50LXNlY3JldC0x

token=987tghjkiu6trfghjuytrghj
```

The client authenticates as it would with a request to the token endpoint, using the same credentials. The authorization server looks up the token value. If it can find it, it deletes it from whatever data store it's using to keep tokens and responds to the client that everything went OK.

```
HTTP 201 No Content
```

Really, that's it. The client discards its own copy of the token and goes on its way.

[10] RFC 7009 https://tools.ietf.org/html/rfc7009

If the authorization server can't find the token, or the client presenting the token isn't allowed to revoke that token, the authorization server responds that everything went OK. Why don't we return an error in these cases? If we did, we'd be inadvertently giving clients information about tokens other than their own. For instance, let's say that we returned an HTTP 403 Forbidden to a client that was trying to revoke a token from another client. In this case, we probably don't want to revoke that token, since that would open up a denial of service attack against other clients.[11] However, we also don't want to tell the client that the token that it has acquired, through whatever means, is valid and could be used elsewhere. To prevent that information from leaking, we pretend that we've revoked the token every time. To a well-behaving client, this makes no difference in function, and to a malicious client we haven't disclosed anything that we don't intend to. We do, of course, still respond appropriately for errors in client authentication, just as we would from the token endpoint.

11.5.2 Implementing the revocation endpoint

We'll now be adding revocation support to our authorization server. Open up `ch-11-ex-5` and edit the `authorizationServer.js` file. We're going to set up our revocation endpoint on `/revoke` on our authorization server, listening to HTTP POST messages. We're going to be importing the client authentication code directly from our token endpoint as well.

```
app.post('/revoke', function(req, res) {
  var auth = req.headers['authorization'];
  if (auth) {
       var clientCredentials = decodeClientCredentials(auth);
       var clientId = clientCredentials.id;
       var clientSecret = clientCredentials.secret;
  }

  if (req.body.client_id) {
       if (clientId) {
              res.status(401).json({error: 'invalid_client'});
              return;
       }

       var clientId = req.body.client_id;
       var clientSecret = req.body.client_secret;
  }

  var client = getClient(clientId);
  if (!client) {
       res.status(401).json({error: 'invalid_client'});
       return;
  }
```

[11] At the risk of complicating matters even more, the details of this specific use case are a bit more subtle because we can now detect that one client has been compromised and had its tokens stolen, and we probably want to do something about that.

```
    if (client.client_secret != clientSecret) {
        res.status(401).json({error: 'invalid_client'});
        return;
    }

});
```

The revocation endpoint takes in one required argument, `token`, as a form-encoded parameter in the body of an HTTP POST, in the same manner as the introspection endpoint. We'll parse out that token and look it up in our database. If we find it, and the client that's making the request is the same as the one that the token was issued to, we'll remove it from the database.

```
var inToken = req.body.token;
nosql.remove(function(token) {
    if (token.access_token == inToken && token.client_id == clientId) {
        return true;
    }
}, function(err, count) {
    res.status(204).end();
    return;
});
```

Whether we remove the token or not, we act as if we have done so and tell the client that everything is OK. Our final function looks like listing 12 in appendix B.

As with introspection, the authorization server also needs to be able to respond to requests to revoke refresh tokens, so a fully compliant implementation will need to check the data store for refresh tokens in addition to access tokens. The client can even send a `token_type_hint` parameter that tells the authorization server where to check first, though the authorization server is free to ignore this advice and check everywhere. Furthermore, if a refresh token is revoked, all access tokens associated with that refresh token should also be revoked at the same time. Implementation of this functionality is left as an exercise to the reader.

11.5.3 Revoking a token

Now we're going to let our client revoke a token. We're going to be revoking tokens in reaction to an HTTP POST to a URL on the client. We've already wired up a new button on the client's homepage to let you access the functionality from the UI. In a production system, you would want this functionality to be protected in order to prevent external applications and websites from revoking your application's tokens without its knowledge (figure 11.3).

We'll start by wiring up a handler for the `/revoke` URL, listening for HTTP POST requests.

```
app.post('/revoke', function(req, res) {

});
```

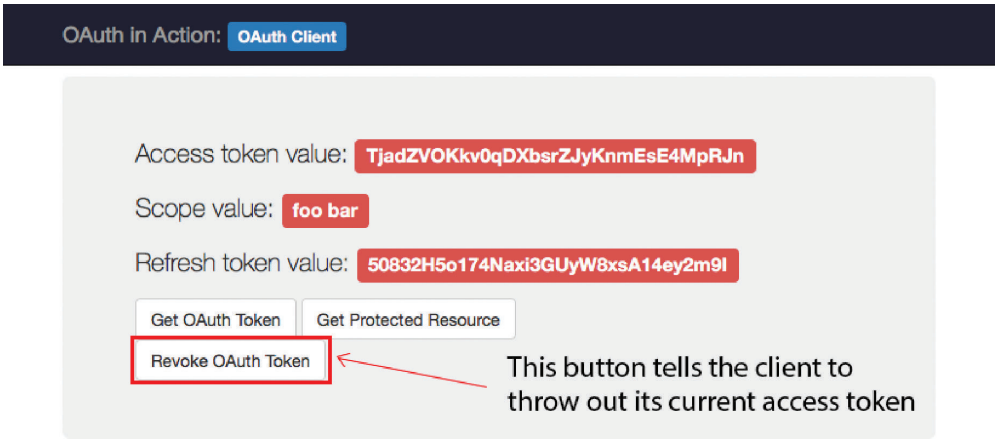

Figure 11.3 The client's homepage with a control to trigger token revocation

Inside this method, we're going to create a request to the revocation endpoint. The client will authenticate using its regular credentials, passed over the HTTP Basic Authorization header, and it will pass its access token as a form parameter in the request body.

```
var form_data = qs.stringify({
   token: access_token
});
var headers = {
   'Content-Type': 'application/x-www-form-urlencoded',
   'Authorization': 'Basic ' + encodeClientCredentials(client.client_id,
   client.client_secret)
};

var tokRes = request('POST', authServer.revocationEndpoint, {
   body: form_data,
   headers: headers
});
```

If the response comes back with a success-class status code, we render the main page of the application again. If it comes back with an error code, we'll print an error for the user. Either way, we throw away the access token, just to be safe on our side.

```
access_token = null;
refresh_token = null;
scope = null;

if (tokRes.statusCode >= 200 && tokRes.statusCode < 300) {
   res.render('index', {access_token: access_token, refresh_token: refresh_
      token, scope: scope});
   return;
} else {
   res.render('error', {error: tokRes.statusCode});
   return;
}
```

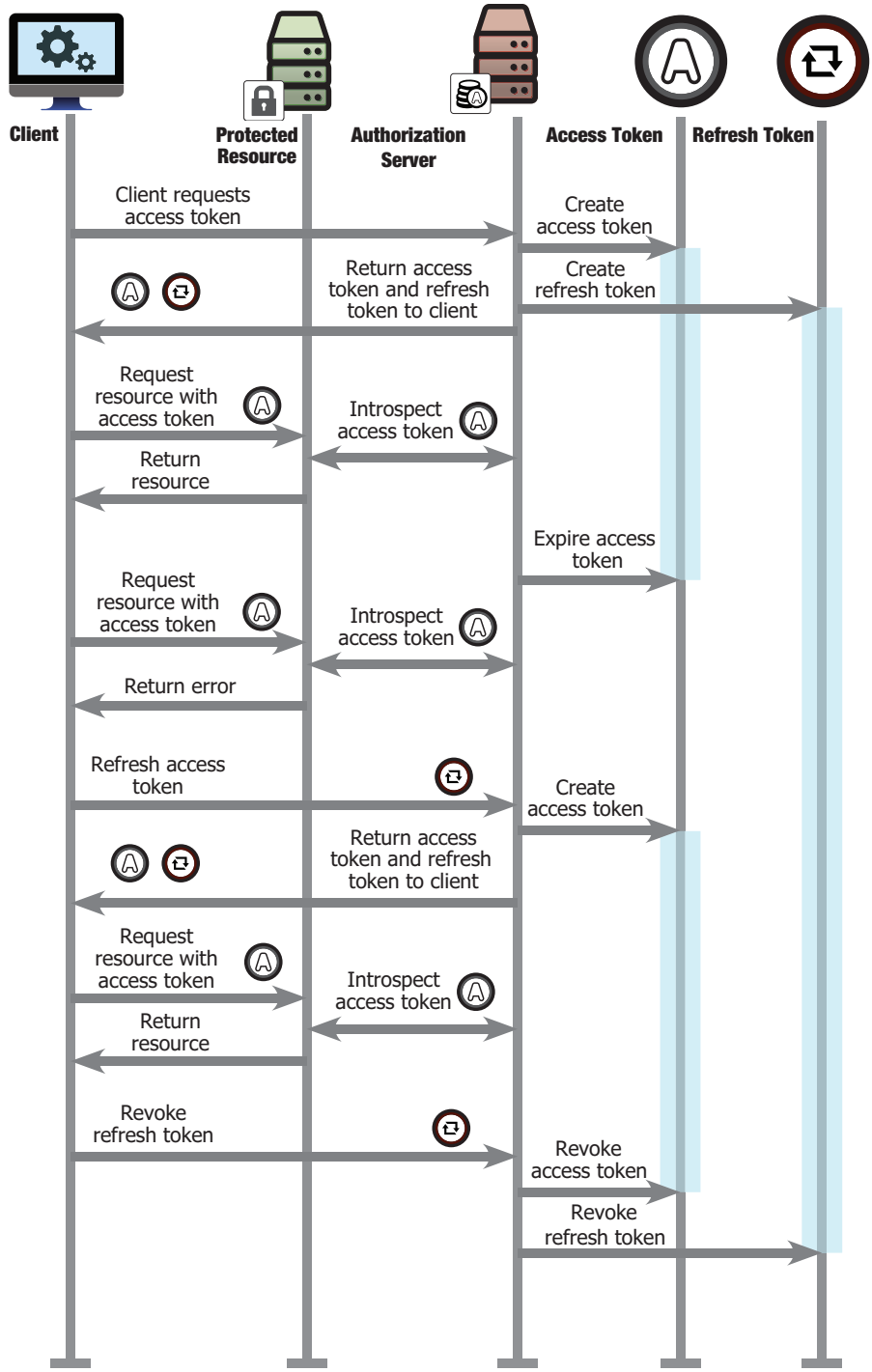

Figure 11.4 The OAuth token lifecycle

The client can request to revoke its refresh token in much the same way. When the authorization server gets such a request, it should throw away all access tokens associated with the refresh token as well. Implementation of this functionality is left as an exercise to the reader.

11.6 The OAuth token lifecycle

OAuth access tokens and refresh tokens have a definite lifecycle. They're created by the authorization server, used by clients, and validated by protected resources. We've also seen that they can be invalidated by a number of factors, including expiration and revocation. Overall, the token lifecycle can be something like what is shown in figure 11.4.

Although this particular pattern is increasingly common, there are many other ways to deploy an OAuth system, such as using stateless JWTs that expire but can't be revoked. However, overall, the general pattern of token use, re-use, and refresh remains the same.

11.7 Summary

OAuth tokens are the central defining component of an OAuth system.

- OAuth tokens can be in any format, as long as it's understood by the authorization server and protected resources.
- OAuth clients never have to understand the format of a token (and shouldn't ever try to do so, anyway).
- JWT defines a way to store structured information in a token.
- JOSE provides methods to cryptographically protect the content of a token.
- Introspection allows a protected resource to query the state of a token at runtime.
- Revocation allows a client to signal the authorization server to discard unwanted tokens after they have been issued, completing the token lifecycle.

Now that you've had a thorough rundown of what OAuth tokens are all about, let's take a look at how clients can be introduced to authorization servers with dynamic client registration.